University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Lab 0 is due this Friday (not graded)
- Recitations start next week
- Homework 1 will be assigned this Friday
- JDB Example will be available on Canvas
- Draft slides and handouts available on Canvas

Recall from previous lecture

runtime of an algorithm =

And after we group statements with the same frequency into blocks

$$= \sum_{all\ blocks} Cost * frequency$$

What is the runtime of ThreeSum?

```
public static int count(int[] a) {
   int n = a.length;
   int cnt = 0;
   for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++) {
         for (int k = j+1; k < n; k++) {
            if (a[i] + a[j] + a[k] == 0) {
               cnt++;
            }
         }
      }
   }
}
```

frequency: $f_0 = 1$
cost: t_0

freq: $f_1 = n$
cost: t_1

$f_4 = x$ (the number of triples that sum to 0 in the input array)
$0 \leq x \leq C(n, 3)$
cost: t_4

$$\frac{(n-3)!6}{6} \quad \frac{6}{2} + \frac{n}{3}$$

cost: t_3

What is the runtime of ThreeSum?

frequency: $f_0 = 1$
cost: t_0

freq: $f_1 = n$
cost: t_1

$f_2 = (n-1) + (n-2) + (n-3) + \ldots + 1$
$= \frac{n-1}{2}(n - 1 + 1) = \frac{n^2}{2} - \frac{n}{2}$
cost = t_2

$f_4 = x$ (the number of triples that sum to 0 in the input array)
$0 \leq x \leq C(n, 3)$
cost: t_4

$f_3 = C(n, 3) = n_{C_3} = \frac{n!}{(n-3)!3!}$
$= \frac{n(n-1)(n-2)(n-3)!}{(n-3)!6} = \frac{n(n-1)(n-2)}{6} = \frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3}$
cost: t_3

Grand total = $\sum_{i=0}^{4} f_i * t_i$

$$= \frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4 x$$

What is the runtime of ThreeSum?

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4x$$

- Remember that $0 \leq x \leq C(n, 3)$
- If x = 0 → best-case runtime

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0$$

- If x = $C(n, 3)$ → worst-case runtime

$$\frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4(\frac{n^3}{6} - \frac{n^2}{2} + \frac{n}{3})$$

Algorithm Analysis

- You see that this analysis can get ugly at times
- Do we really need to consider all these terms and constants?
  - The answer is No!

Enter Asymptotic Analysis

Algorithm Analysis

- Determine *resource usage* as a function of *input size*
  - e.g., *n,* in 3-sum, the length of the array size, is the input size
  - We already did that for ThreeSum
- Measure ***asymptotic*** performance
  - Performance as input size increases to infinity

Asymptotic performance

Focus on the **order of growth** of functions, not on exact values

Asymptotic performance

Order of growth captures how fast the function value increases when the input increases; in particular, for a function *T(n)*

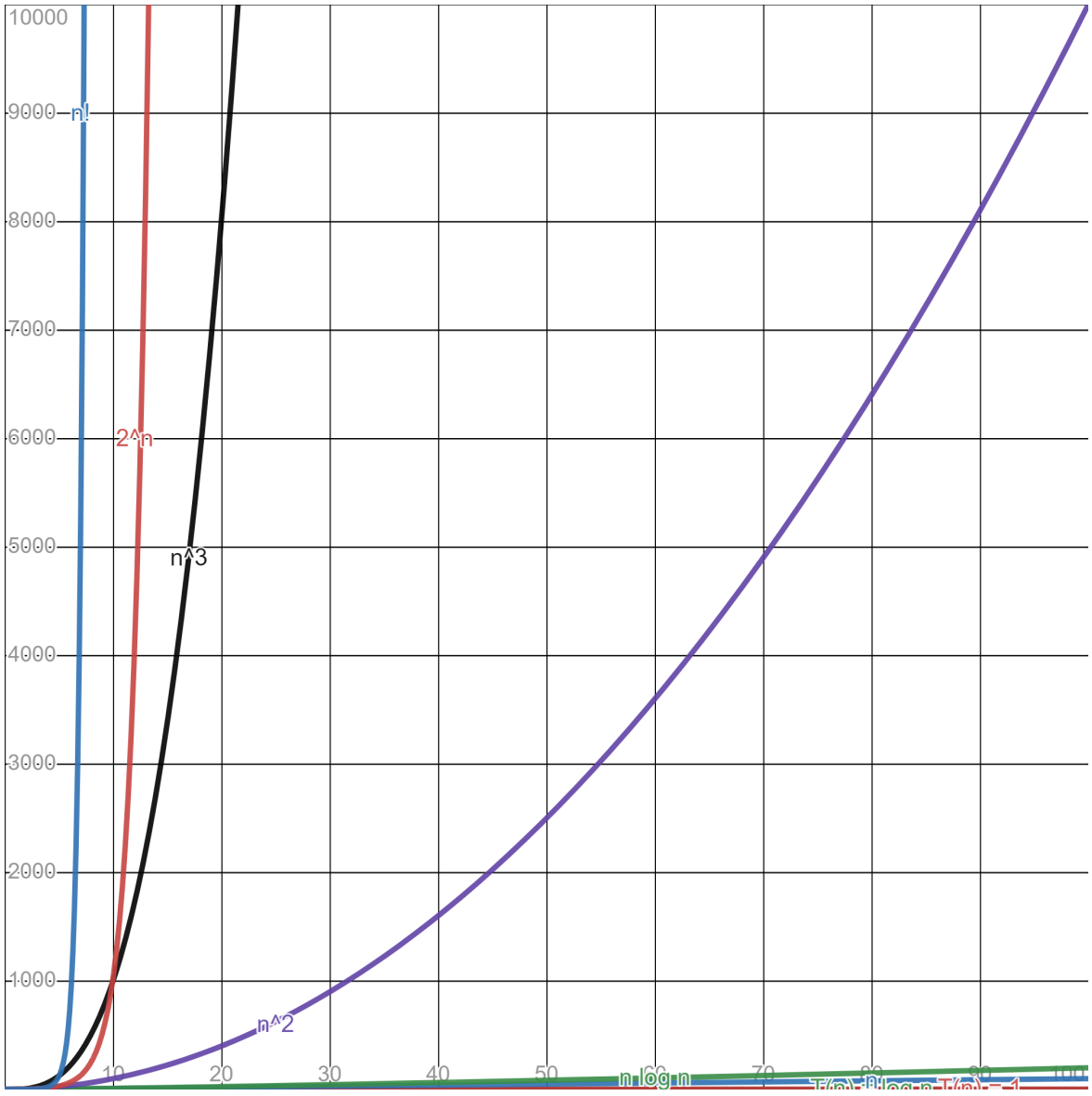
- When *n* doubles, does *T(n)* essentially
  - stay constant
  - increase by a constant
  - double as well
  - quadruple (**x**4)
  - increase eightfold (**x**8)
  - … ?
- When *n* increases by 1, does *T(n)* essentially
  - double
  - increase n-fold (**x**n)
  - … ?

Asymptotic performance

We don’t care as much about the exact value of $T(n)$

Common orders of growth in Algorithm Analysis

- Constant - 1
- Logarithmic - log n
- Linear - n
- Linearithmic - n log n
- Quadratic - n^2
- Cubic - n^3
- Exponential - 2^n
- Factorial - n!



Side note

What does $log_2 n$ really mean?

$log_2 n$ is the number of times n can be divided by 2 before until we reach 1 or less

Side note

Why do we use *T(n)* instead of *f(x)?*

T stands for Time, or running time

Using *n* signifies that the input is a positive integer

Order of growth of runtime functions

- For runtime functions, is it better to have a function with a high order of growth or a low order of growth?
  - low order of growth means when input size increases, the value of the runtime won't increase by much
  - This means a fast algorithm
  - So, we want a low order of growth function for runtime

Quick algorithm analysis

- How can we determine the order of growth of a function?
  - Ignore lower-order terms
  - Ignore multiplicative constants
- Example: polynomial functions
  - $\mathrm{T(n)} = 5n^3 + 53n + 7$
  - Terms: $5n^3, 53n, 7$
  - $5n^3$ is of order 3, $53\ n$ is of order 1
  - what is the order of the term 7?

Example

$$5n^3 + 53n + 7 \rightarrow n^3$$

- Warning: this is a simplification
  - It works for most of the algorithms in this course
- In some cases, it is difficult to determine the highest-order term
- In some cases, the constant factors play a significant role
  - e.g., small or medum-size input and large constant factors

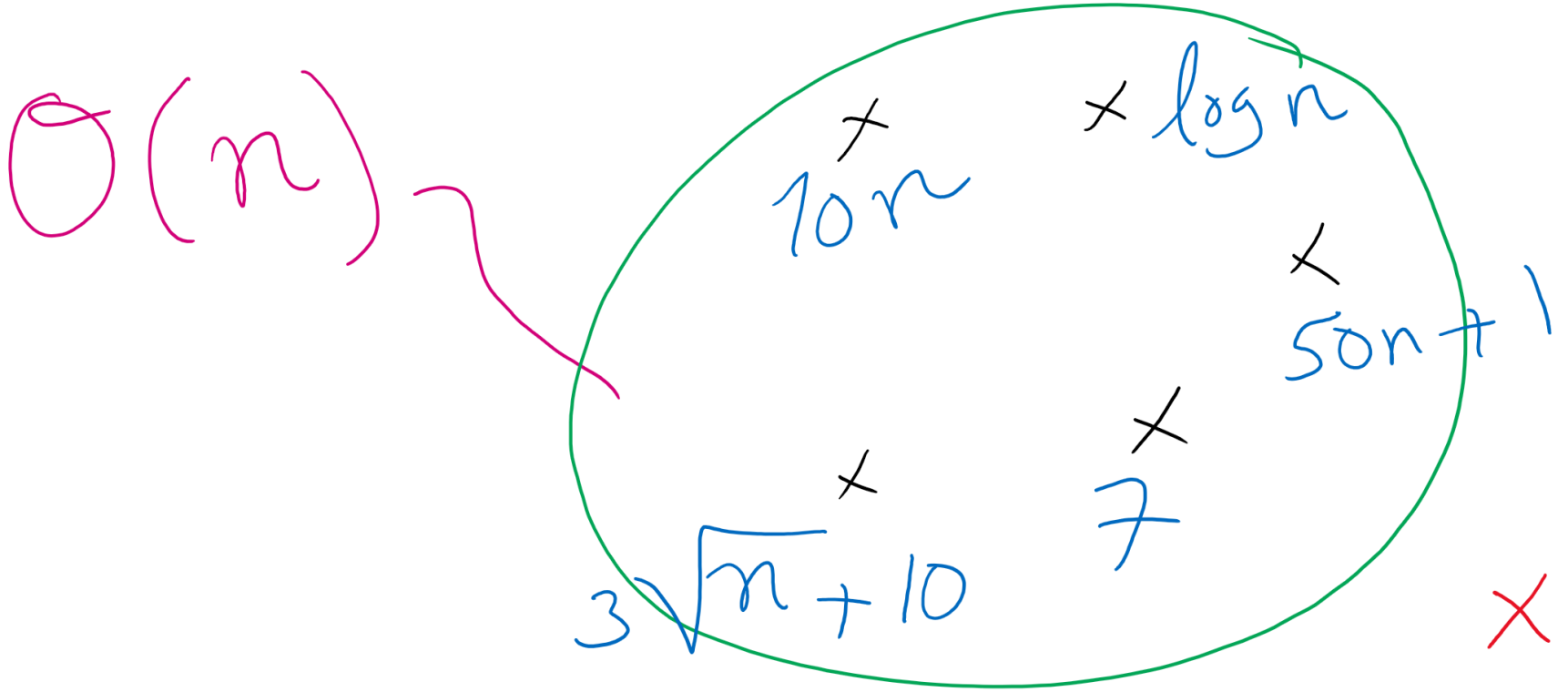
But …

- Can we say $5n^3 + 53n + 7 = n^3$?
- No! We need a mathematical notation
- $5n^3 + 53n + 7 = O(n^3)$
  - Read as Big O of n^3
- It means the order of growth of $5n^3 + 53n + 7$ is no more than ($\leq$) the order of growth of n^3

Notations

- May also see:
  - $f(x) \in O(g(x))$ or
  - $f(x) = O(g(x))$
- used to mean that f(x) is O(g(x))
- Same for the other functions

Notations



$10n \in O(n)$

$\log n \in O(n)$

$n^2 \notin O(n)$

The Big O Family

- O roughly means ≤
  - Big O
- o roughly means <
  - Little O or O-micron
- Ω roughly means ≥
  - Big Omega
- ω roughly means >
  - Little Omega
- Θ roughly means =
  - Theta
- Relationships are between orders of growth, not between exact values!

Asymptotic analysis approximations

- How can we determine the order of growth of a function?
  - Ignore lower-order terms
  - Ignore multiplicative constants
- Would it matter if the frequency of a statement is *n* or *n+1?*
  - *No!*
- Would it matter if it is *n* or *2n?*
  - *No!*
- Would it matter if it is 2^n or 2^{2n}?
  - Yes! Why?

A couple useful approximations under Asymptotic Analysis

$\sum$ arithmetic series:

$$1 + 2 + 3 + 4 + \cdots + n$$

$$= \#\text{terms} \left(\frac{\text{first term} + \text{last term}}{2} \right)$$

$$= \Theta(\#\text{terms} * \text{largest term})$$

$\sum$ geometric series:

$$1, 2, 4, 8, 16, \cdots$$

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \cdots$$

$$= \Theta(\text{largest term})$$

Let's go back to our ThreeSum Algorithm

- We know that

$$T(n) = \frac{t_3}{6}n^3 + (\frac{t_2}{2} - \frac{t_3}{2})n^2 + (\frac{t_3}{3} - \frac{t_2}{2} + t_1)n + t_0 + t_4x$$

- What is the order of growth of T(n)?
  - $T(n) = O(n^3)$

Let's go back to our ThreeSum Algorithm

- Assuming that definition…
  - Is ThreeSum $O(n^4)$?
  - What about $O(n^5)$?
  - What about $O(3^n)$??
- If all of these are true, why was $O(n^3)$ what we jumped to to start?

Another mathematical notation

Tilde approximation (~)

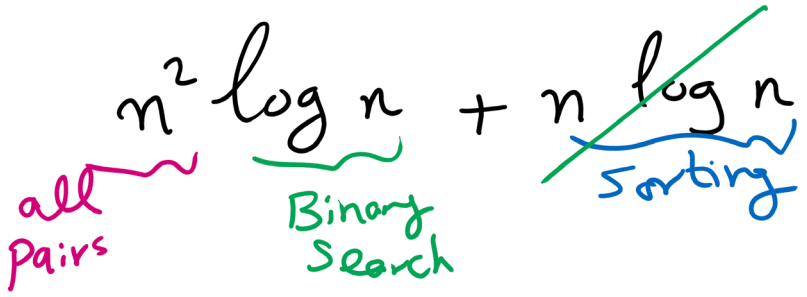
- Same as Theta but keeps constant factors
- Two functions are Tilde of each other if they have the same order of growth and the same constant of the largest term

$$5n = \Theta(5{,}000{,}000{,}000\ n)$$

$$\neq \sim 5{,}000{,}000{,}000\ n$$

A faster algorithm for 3-sum

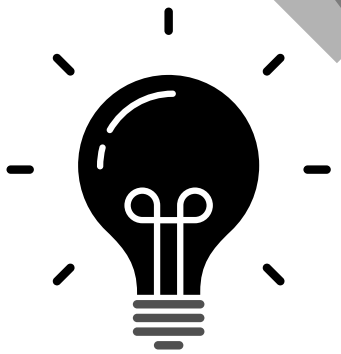
- What if we sorted the array first?
  - For each pair of numbers, binary search for the third one that will make a sum of zero
    - e.g., a[i] = 10, a[j] = -7, binary search for -3
    - Be careful not to use the same number twice
- What is the runtime?
  - Still have two for-loops, but we replace the third with a binary search
    - What if the input data isn't sorted?
  - What about the sorting time?



The 3-sum problem: can we do better?

- There is an O(n^2) algorithm
  - Idea 1: use hashing to find the third number
  - Idea 2: for each number, find the missing pair of numbers in linear time
- There is also an O(n log n) algorithm under special cases
- **Unsolved problem**: Is there a general O($n^{2-\varepsilon}$) algorithm for some $\varepsilon > 0$?

Bonus Alert!



- Modify ThreeSum to work correctly with duplicates in the input
- Write an $O(n^2)$ solution to the 3-sum problem

Send your solution using Piazza in a private message to all instructors labeled with the “bonus” tag

Another problem: Boggle

- Given a 4x4 board of letters, find all words with at least 3 adjacent letters
- Adjacent letters are horizontally, vertically, or diagonally neighboring
- Any cube in the board can only be used once per word
  - but can be used fo multiple words

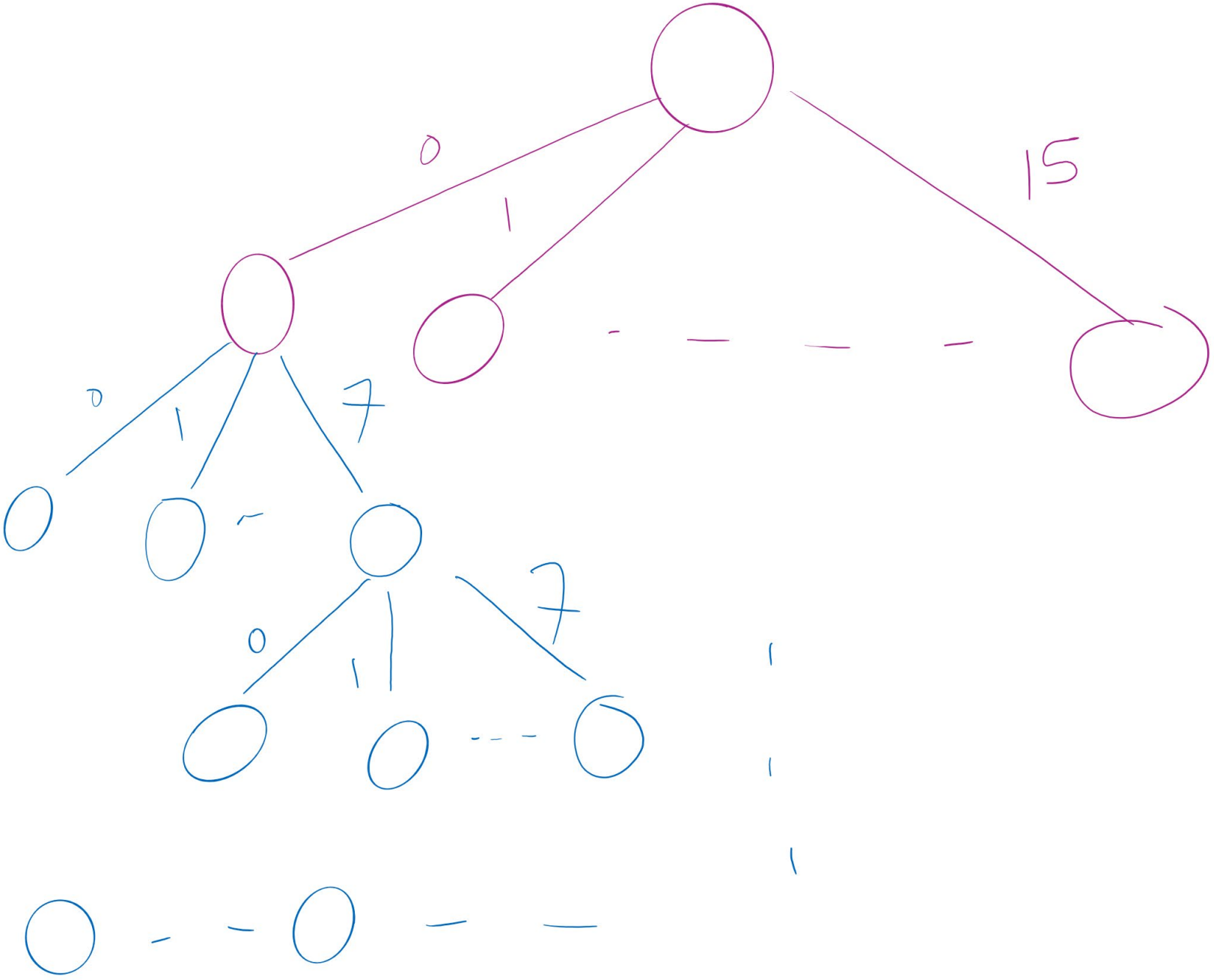
Recursing through Boggle letters

- Have 16 different options to start from
- Have 8 different options from each cube
  - From B[i][j]:
    - B[i-1][j-1]
    - B[i-1][j]
    - B[i-1][j+1]
    - B[i][j-1]
    - B[i][j+1]
    - B[i+1][j-1]
    - B[i+1][j]
    - B[i+1][j+1]

Boggle Game: Decisions and Choices

- For the first decision (which cube to start from), how many possible choices do we have?
- Starting from a cube, the next decision to make is which adjacent cube to move to.
  - There are at most 8 possible choices to choose from
- There is a decision to make at each cube that we go through
  - A maximum of 15 decisions
  - Some decisions may be trivial if the number of choices is 0

Search Space



First decision

Are all 16 choices valid?

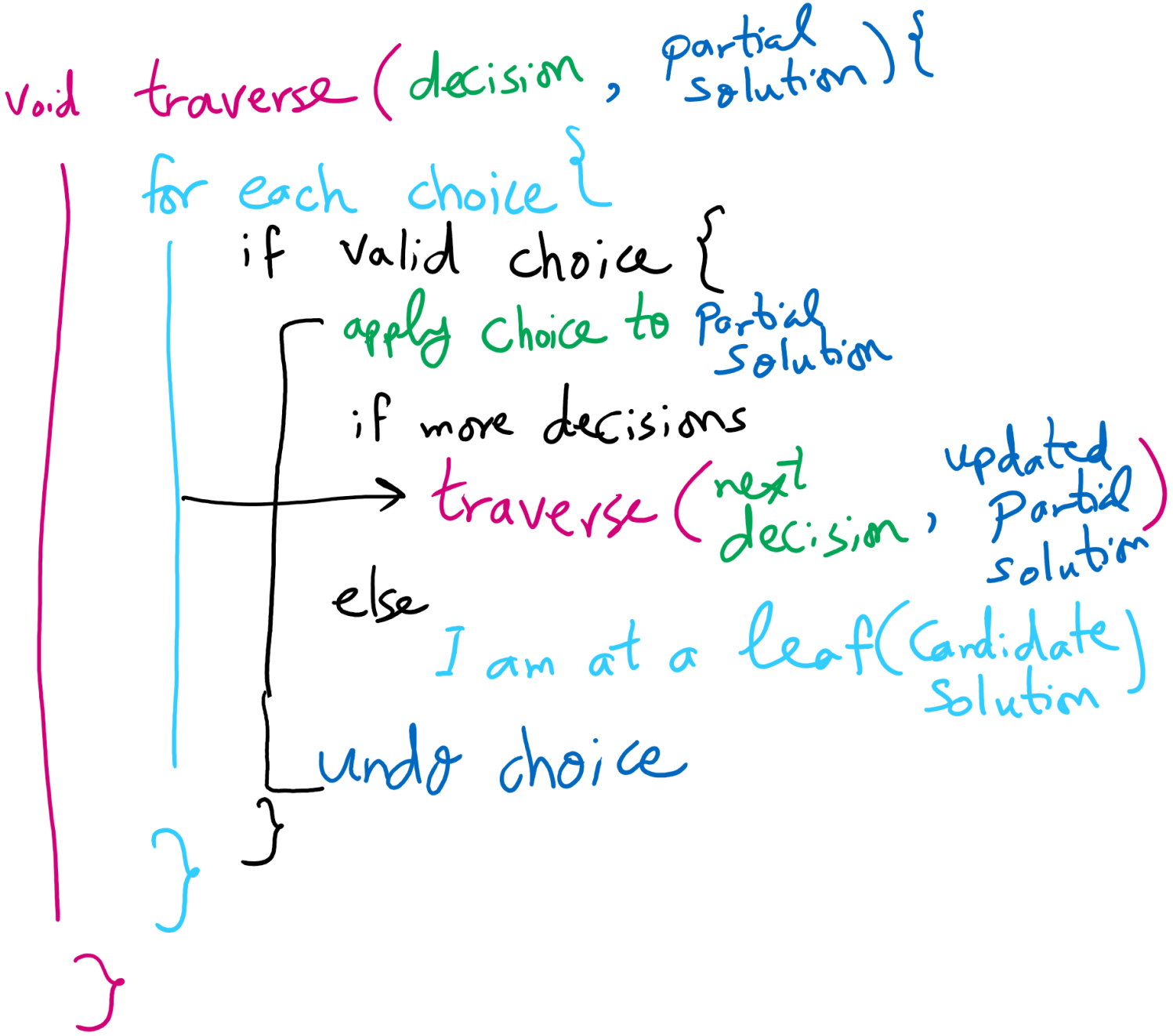
For next decisions, are all 8 chices valid?

- Can't go past the edge of the board
- Can't reuse cells
- Each letter added must lead to the prefix of an actual word
  - check the dictionary as we add each letter, if there is no word with the currently constructed string as a prefix, don't go further down this way
  - Practically, this can be used for huge savings
  - This is called pruning!

Then what?

- How can we code up such exhaustive search with pruning?
- The algorithm is called backtracking.

Backtracking Algorithm



Backtracking Framework

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

Application to Boggle Game

void traverse(current decision, ….

Which cube are we at:

row number (0..3) and column number (0..3)

Backtracking Framework

```
void traverse(….., partial solution) {

    ….

}
```

The letters that we have seen so far

Backtracking Framework

```
void traverse(….) {
   for each choice at the current decision {
      …
    }
}
```

We have 8 choices (except for the first decision, which has 16 choices)

- From B[i][j]:
  - B[i-1][j-1]
  - B[i-1][j]
  - B[i-1][j+1]
  - B[i][j-1]
  - B[i][j+1]
  - B[i+1][j-1]
  - B[i+1][j]
  - B[i+1][j+1]

Backtracking Framework

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         ...
      }
   }
}
```

Invalid choices send us outside the board, reuse a cube, result in a non-prefix

Backtracking Framework

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         ...
      }
   }
}
```

Append the letter to the partial solution and mark the cube as used

Backtracking Framework

```
...
if partial solution a valid solution
   report partial solution as a final solution
...
```

If the partial solution is a word in the dictionary

Backtracking Framework

```
...
if more decisions possible
...
```

A decision is possible if partial solution is a prefix of a word in the dictionary

Backtracking Framework

```
...
traverse(next decision, updated partial solution)
...
```

a recursive call.

next decision is the next cube down the direction that we chose

Backtracking Framework

...

undo changes to partial solution

...

Remove the last letter that we appended from partial solution and mark cube as unused

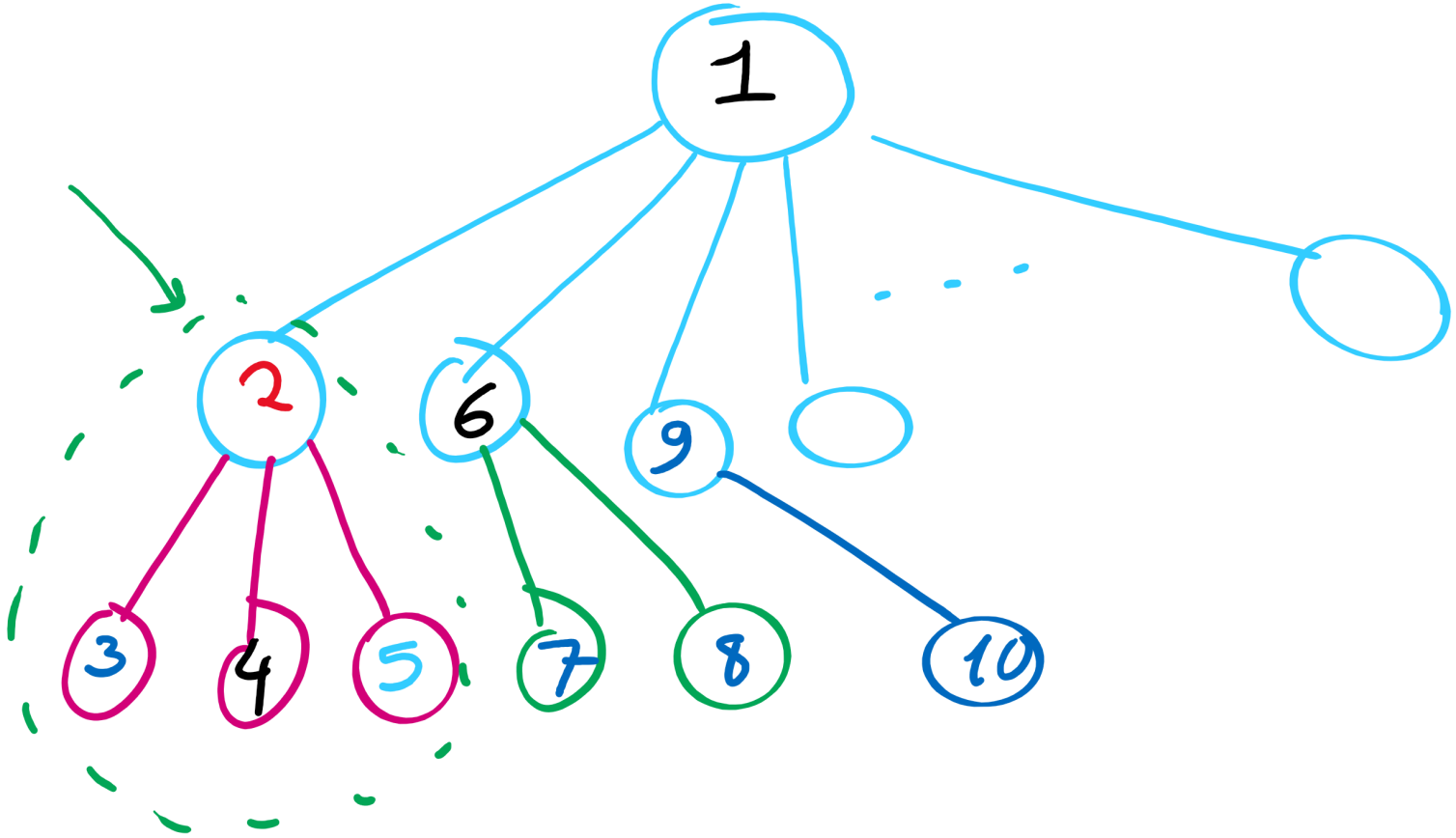
Backtracking Framework

```
void traverse(current decision, partial solution) {
   for each choice at the current decision {
      if choice is valid {
         apply choice to partial solution
         if partial solution a valid solution
            report partial solution as a final solution
         if more decisions possible
            traverse(next decision, updated partial solution)
          undo changes to partial solution
      }
   }
}
```

You will implement this algorithm in Lab 1 next week!

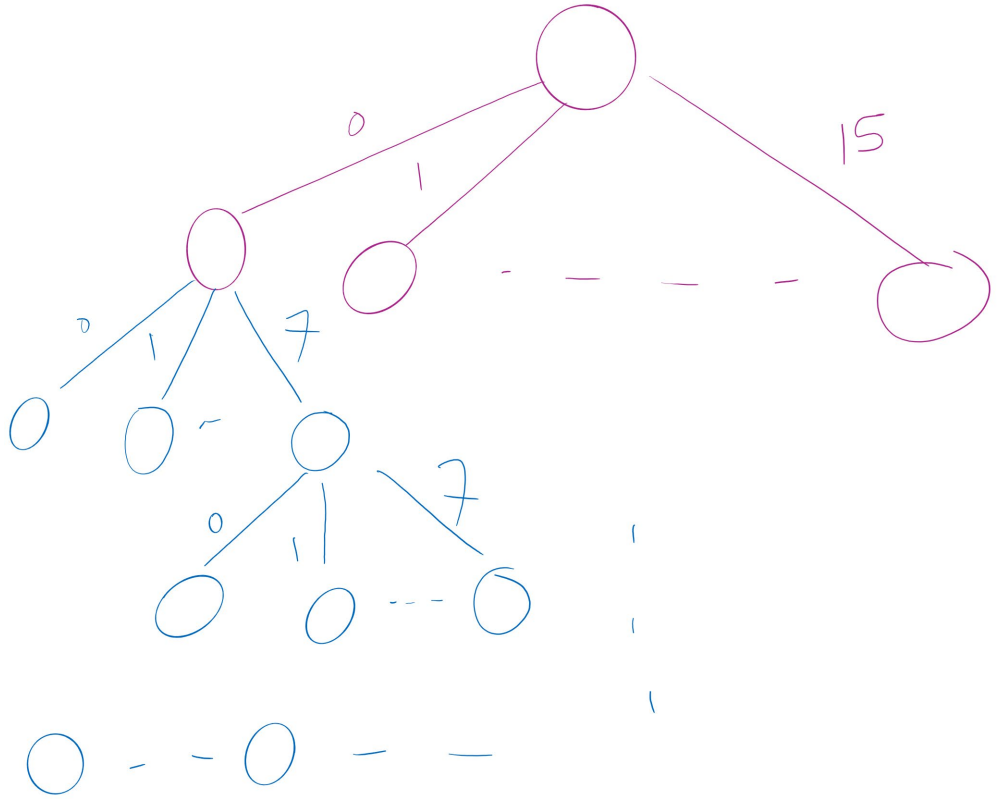
Search Tree Traversal Order in Backtracking

- Each node (circle) corresponds to a recursive call
- The runtime cost per node is the cost of all statements except the recursive call



How many nodes are there?

$$\#\text{leaves} = \Theta(\#\text{nodes}) = \Theta\left(\left(\#\text{ branches per node}\right)^{\#\text{ levels} - 1}\right)$$



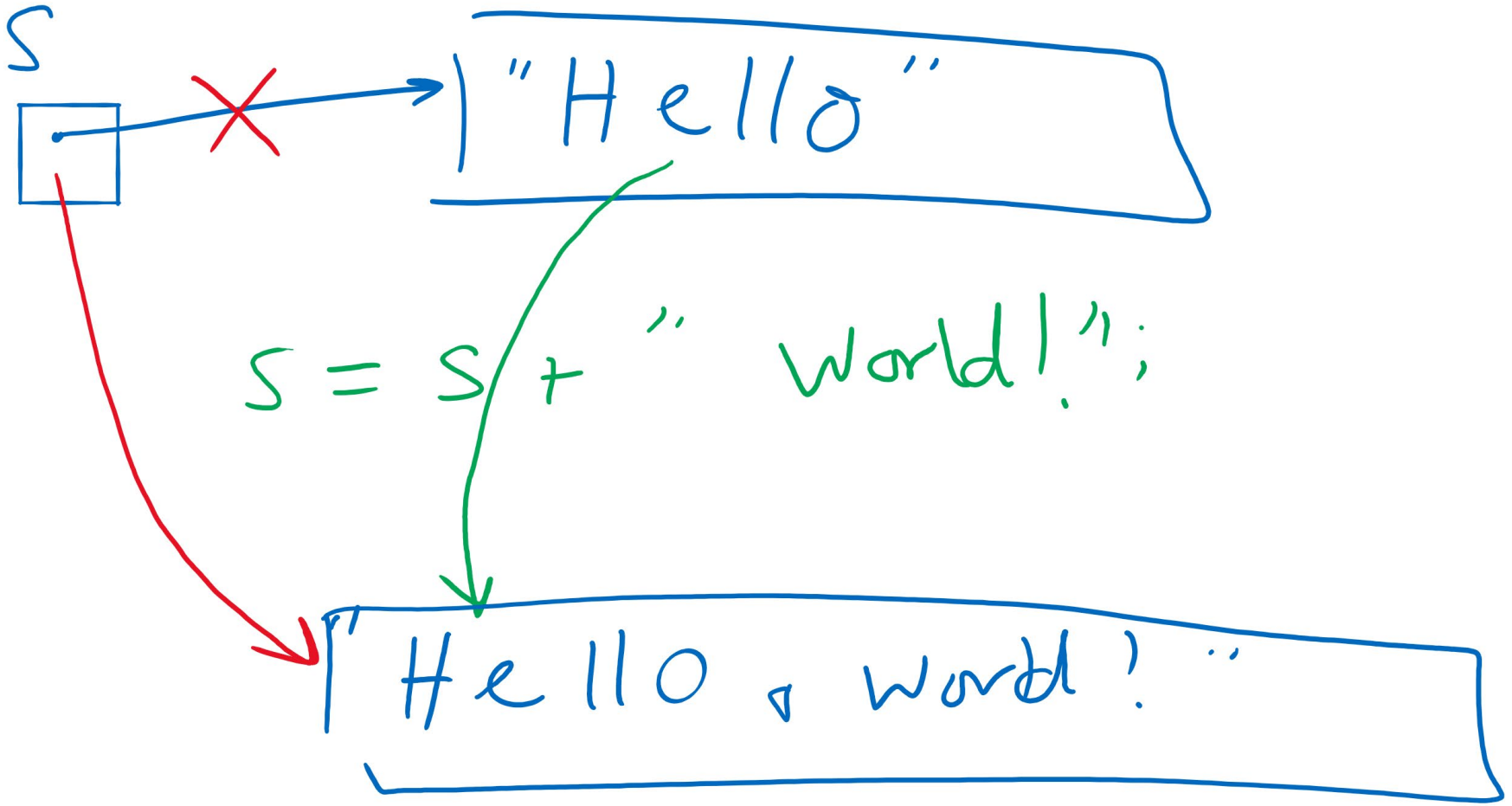
What is the running time?

- Can't really use the frequency and cost technique because of the recursive call
- We can use the number of nodes in the search tree as a lower bound on the worst-case runtime
  - worst-case runtime of Boggle = $\Omega(8^n)$, where *n* is the board size
  - if the non-recursive work is constant, then
    - worst-case runtime = $O(8^n)$
  - How can we make it constant?

Implementation concerns with Boggle

- Constructing the words over the course of recursion will mean building up and tearing down strings
  - Moving forward adds a new character to the current word string
  - Backtracking removes the most recent character
  - Basically pushing/popping to/from a string stack
- Push/Pop stack operations are generally $\Theta(1)$
  - Unless you need to resize, but that cost can be amortized
- Java Strings, however, are *immutable*
  - s = new String("Here is a basic string");
  - s = s + " this operation allocates and initializes all over again";
  - Becomes essentially a $\Theta(n)$ operation
    - Where n is the length of the string

Concatenating to A String Object



StringBuilder to the rescue

- append() and deleteCharAt() can be used to push and pop
  - Back to Θ(1)!
    - Still need to account for resizing, though…
- StringBuffer can also be used for this purpose
  - Differences?